Applications of Artificial Intelligence

Sebastian Iwanowski
FH Wedel

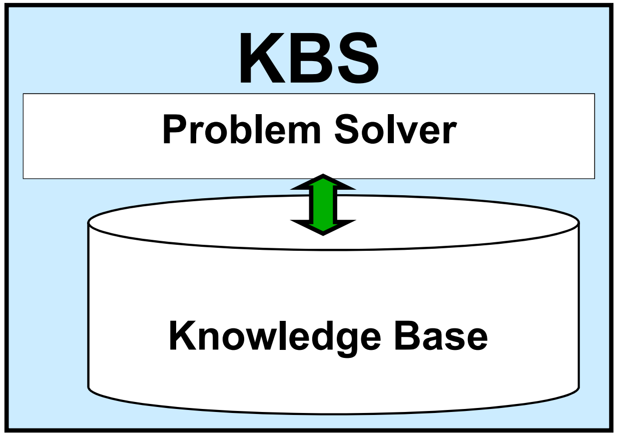
Chapter 3:
Algorithmic Methods of AI

Search Strategies

Relevance of search strategies for logic problems:

Search for a solution of the satisfiability problem

Relevance of search strategies for knowledge-based systems:



The problem solver nearly always has to solve a satisfiability problem for constraints of the knowledge base!

➔ All problem solvers search

Example for a knowledge-based search engine: PROLOG

PROLOG is knowledge-based:

- **Knowledge base**

  Facts and rules, dynamically extensible

- **Inference engine („Problem Solver“)**

  deriving facts and rules automatically

- **Dialog component**

  Input: Query
  Output: yes / no, specification of unification applied in case of success,
  write as a „side effect“

  Yes: The predicate of the query can be concluded from knowledge base.
  No: The predicate of the query cannot be concluded from knowledge base.
  No does not imply that it can be concluded that the predicate is false.

Application: Class Scheduling

Given finite sets Courses, **R**ooms, **T**ime slots

Task: Generate an injective (one-to-one) function C → RxT

Strict Constraints (must be fulfilled in any case):

- **Certain courses must not take place at the same time.**
- **For some courses, certain time slots are not admitted.**
- **For some courses, certain rooms are not admitted.**

Soft constraints (may be violated):

- **Certain courses should not take place at some times.**
- **Certain courses should take place successively.**
- **Certain courses should not take place on the same day.**

Optimisation function:

- **fewest violations of soft criteria**
- **fewest free periods for certain study programmes**
- **most uniform distribution on different days for ...**

Application: Traveling Salesman Problem (TSP)

Given: Graph with node set V and weighted edges between the nodes

Task: Find a round trip traversing the graph edges reaching each node at least once.

Constraints:

- **Only edges of the graph are to be used.**

Optimisation function:

- **Minimise the global edge costs !**

Generalisation in logistic applications:

Constraints:

- **Load and destribute goods obeying capacity restrictions !**
- **Consider time windows in which delivery may take place !**

Soft criteria (may be violated):

- **Certain edges have to be avoided.**
- **Certain time windows are unfavourable.**

Application: Shortest Path Problem

Given: Graph with node set V and weighted edges between the nodes

Task: For two selected nodes S and T, find a path through the graph.

Constraints:

- **Only edges of the graph are to be used.**

Optimisation function:

- **Minimise the global edge costs !**

Generalisation in transport applications (public or individual):

Constraints:

- **Edge costs depend on the time used.**
- **Travelors are subject to individual contraints that may value certain edges in a different way or make them even unusable.**

Soft criteria (may be violated):

- **Certain edges have to be avoided**
- **Certain time windows are unfavourable**

Constraint Satisfaction Problem (CSP)

<u>Specification of a CSP:</u>

- **set of variables**
- **domains of definition**
- **constraints: relations between variables (strict or soft)**
  (nomally, equations or inequalities)
- **optimisation criterion**
  (normally, a real-valued function on the variables which has to be minimised or maximised)

<u>valid solution:</u>

assignment of values to all variables such that all strict constraints are satisfied

<u>optimal solution:</u>

valid solution optimising the optimisation criterion

<u>Constraint Solver</u> are programmes which find a valid or even optimal solution for a given CSP automatically.

Traversing search graphs

<u>1. search method:</u> Find a global solution via partial solutions

- **Node: describes state in search domain**
  - **<u>State:</u> Assigning values to variables**

    Each state has got an evaluation.
- **Edge: transition of a state into a subsequent state**

  (usually feasible in one direction only)
  - **<u>Subsequent state:</u> Assign a value to a new variable**
    keeping the values for the already assigned variables
- **Initial node: initial state**

  (is always unique)
  - **<u>Initial node:</u> No variable has got a value.**
- **Final node: final state wanted (problem solution)**

  (several ones are admissible)
  - **<u>Final node:</u> All specified variables have got admissible values.**

Traversing search graphs

<u>Different search goals are possible:</u>

1) **Find some solution or detect that there is none.**
2) **Find further solutions or detect that there are none.**
3) **Find all solutions.**
4) **Find an optimal solution or at least a rather good one.**

- **<u>Expansion</u> of a node: Compute all subsequent resp. adjacent nodes**

Different search strategies differ in:

Which node has to be expanded next?

Special case:

- **Search graph is a search tree**

(makes the path from initial node to each final node unique)

Example for search trees in CSP

Constraint system:

```
1) (2 < x < 4)
2) (0 < y < 6)
3) (x + y > 7)
4) (x · y < 10,5)
```

Domain of definition for valid solutions:

$x,y \in \mathbf{Q}$,
at most k positions after the decimal point

Optimisation criterion:

Minimise $|y – x|$

Search tree:

- Each node has got fixed x and y values, nodes may be valid or not valid, for each node there is a unique optimal value.
- In level i, each x value has got i entries after the decimal point, the y value is minimum according constraint 3).

Expansion strategies:

- Only valid nodes may be expanded.
- The rightmost valid node on the next level is expanded.
- ...

Example for search trees in CSP

Constraint system:

```
1) (2 < x < 4)
2) (0 < y < 6)
3) (x + y > 7)
4) (x · y < 10,5)
```

Domain of definition for valid solutions:

x,y ∈ **Q,**
at most k positions after the decimal point

Optimisation criterion:

Minimise |y – x|

for bounded k:

- finite search space
- several valid solutions
- always 1 optimal solution

for unbounded k:

- infinite search space
- infinitely many valid solutions
- no optimal solution

Uninformed Search Strategies

In general, only *blind (uninformed) search* is possible:

There is no information about good search search directions (the target is only recognised on arrival)

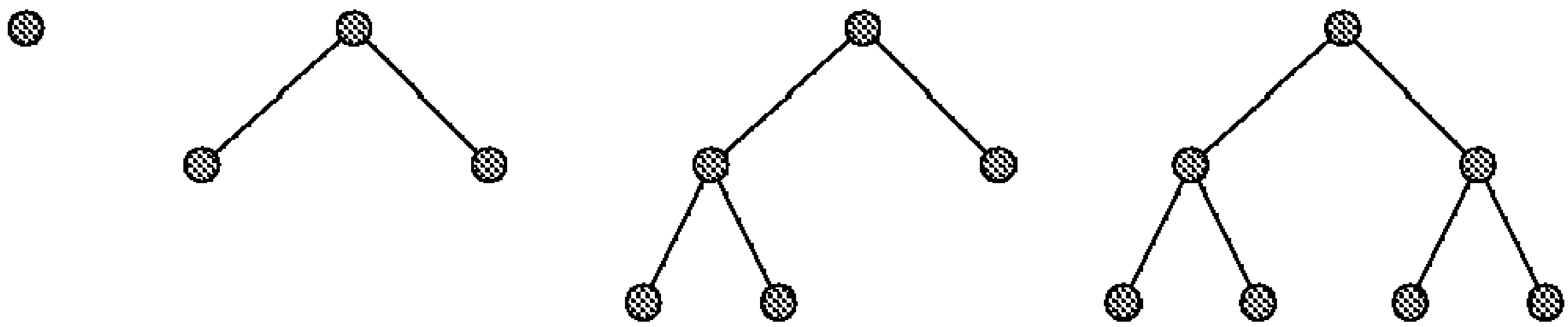
The most important search strategies:

1. breadth first search
2. depth first search
3. best first search

Weitere Infos zum Thema Suchen: Seminarvortrag und Ausarbeitung von Sven Schmidt, SS 2005, Nr. 4
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

Uninformed Search Strategies

breadth first search:



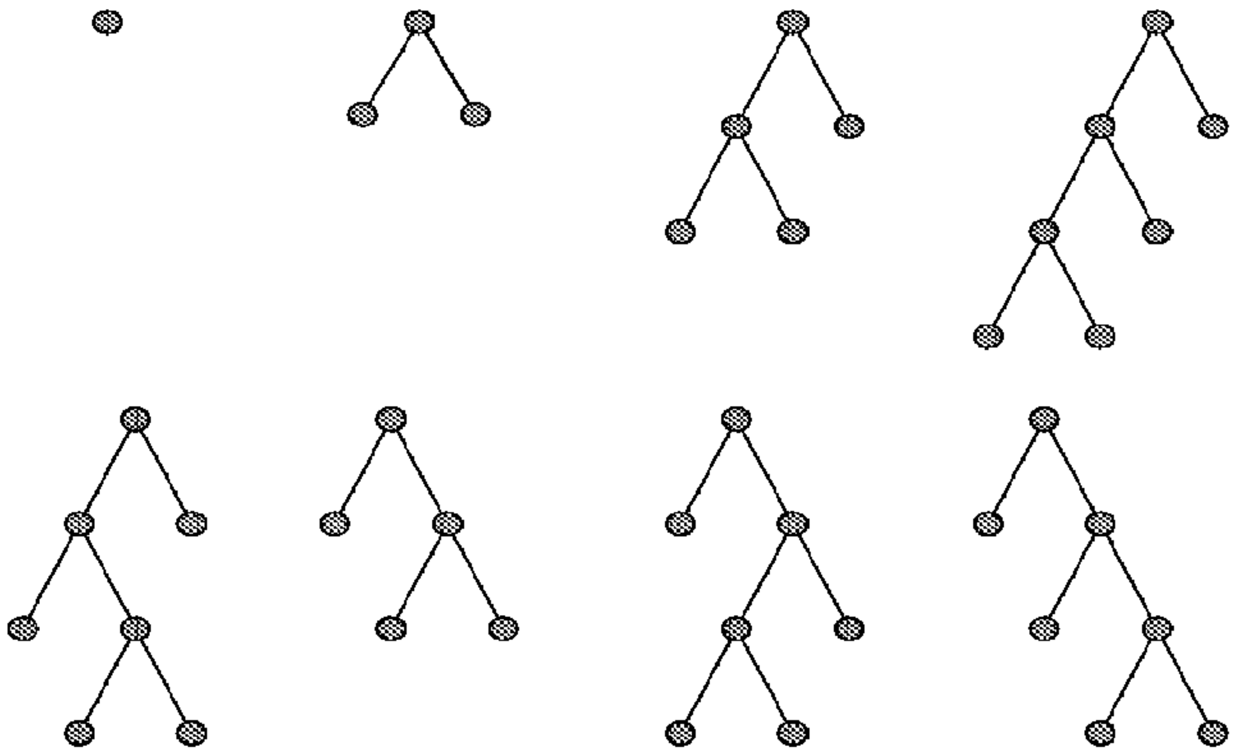
problem size: depth of search tree

Exponential time and space

for AI search procedures not relevant in most cases

Uninformed Search Strategies

depth first search:



problem size: depth of search tree

Exponential time

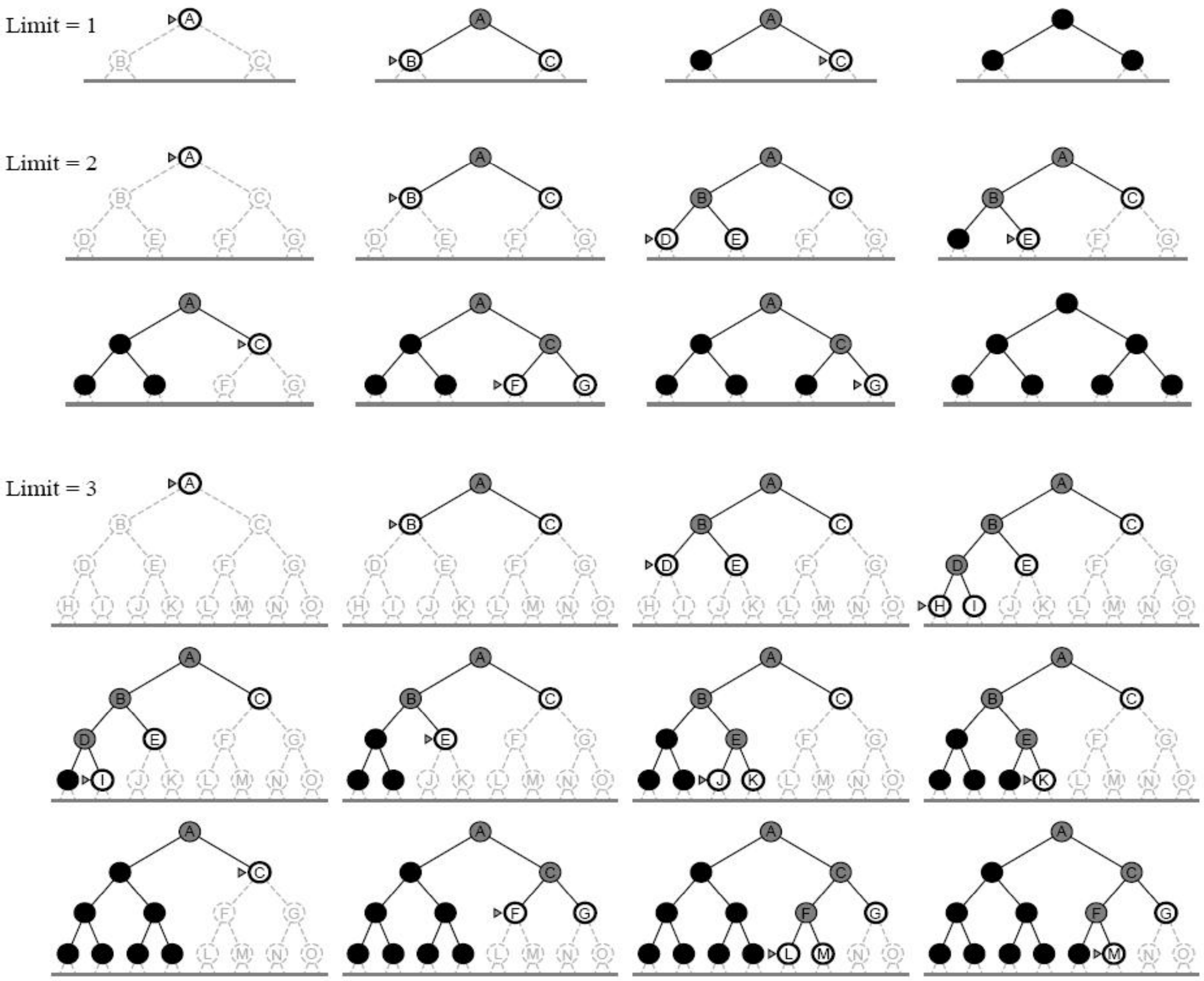
Linear space

The „normal case“ for standard AI procedures

Uninformed Search Strategies

bounded depth first search:

- Execute depth first search only up to limited search level.
- If not successful, increase limit for search level and start depth first search again.



Uninformed Search Strategies

best first search:

- Additional information: Evaluation label for the nodes.
- Search target: Find the best solution first (and the others later).
- Expand the node with best evaluation first.

→ ***Mixture of depth first and breadth first searches***

In the *worst case* this is no better than breadth first search:

Exponential effort for time and space

Problem size:

Depth of search tree

For good evaluation functions, *the avarage case* is much better!

For special problems, even the worst case is much better:

<u>Example:</u> Special case „Shortest Path Problem“:

Dijkstra‘s algorithm (quadratic effort for time, linear for space)

Problem size: Number of nodes

Uninformed Search Strategies

Dijkstra‘s algorithm for weighted graphs

(special case of best first search)

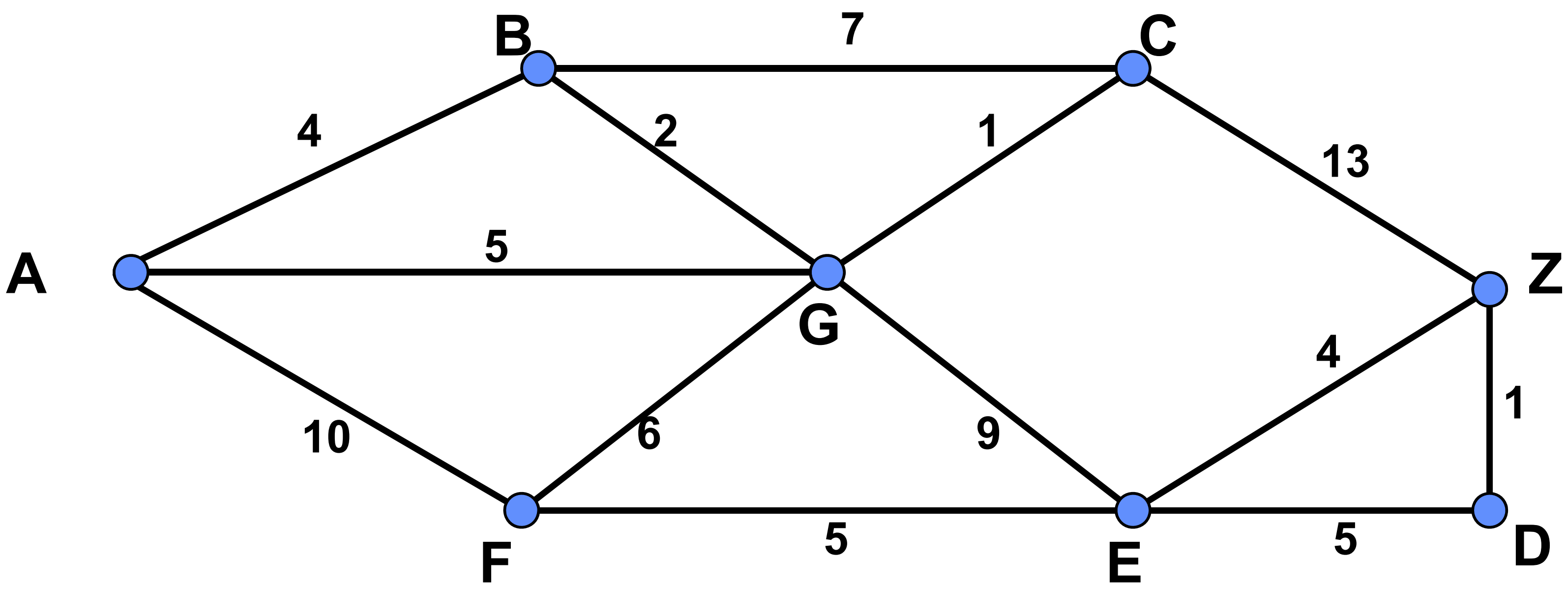
For all edges (u,v) there is a weight function:
length (u,v) := length of an edge from node u to node v

Requirement for edge weights: All lengths have to be nonnegative.

Algorithm for the search of a path from A to B having minimal global edge length:

- Put A into the set **`Done`**. Label A by *distance*(A) := 0.
  Put all other nodes into the set **`YetToCompute`**.
  Label all neighbors N of A by *distance* (N) := *length* (A,N)
  and all othe nodes by *distance* (V) := ∞.
- Repeat:
  Choose node V from **`YetToCompute`** with minimum *distance* (V)
  and shift V to the set **`Done`**.
  Update all neighbors N of V that are still in **`YetToCompute`**:
  distance (N) := min {*distance* (N), *distance* (V) + *length* (V,N)}.
  until V = B

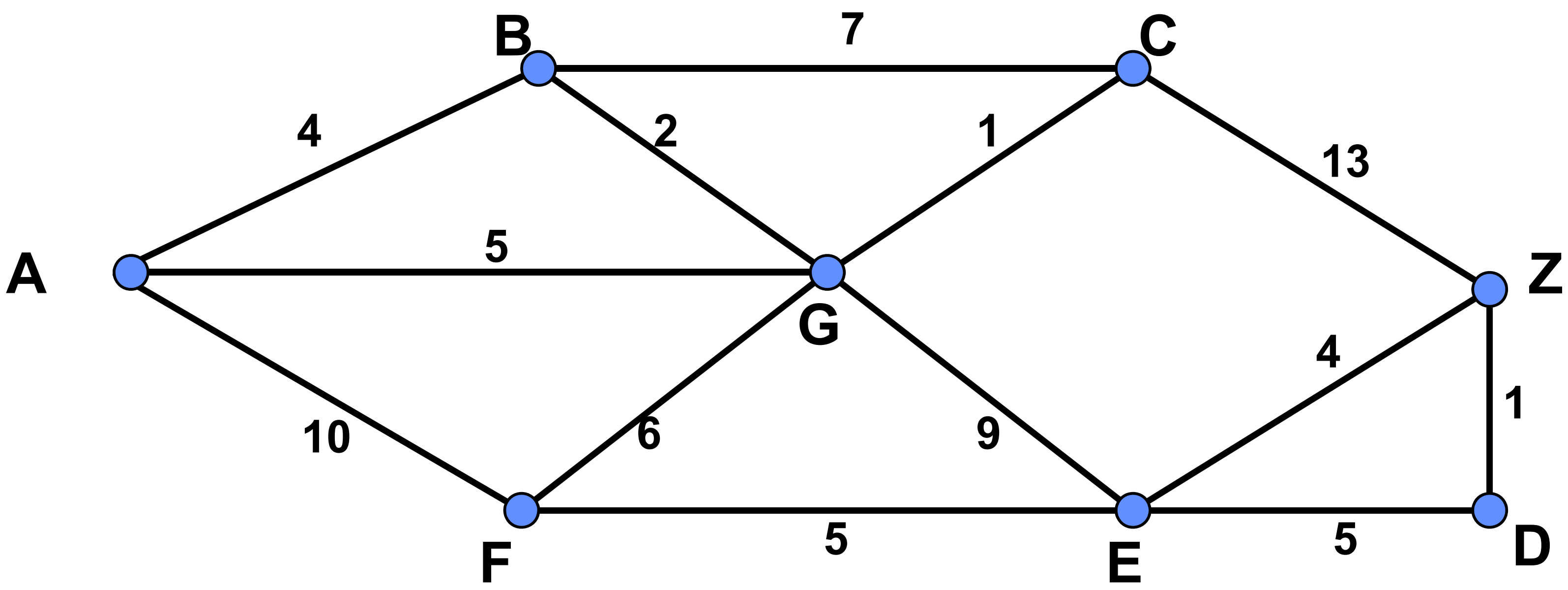
Example for Dijkstra‘s algorithm



Shortest path from A to Z: A → F → E → Z (17 units)

Animation dieser Aufgabe und weitere Infos zum Algorithmus von Dijkstra:
Seminarvortrag und Ausarbeitung von Alex Prentki, WS 2004, Nr. 14
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/WS2004/SeminarMC.html

Example for Dijkstra‘s algorithm



Shortest path from G to Z: G → E → Z (13 units)

Node (distance from G, direct predecessor):

A(5,G)		A(5,G)		$\boxed{\text{A(5,G)}}$						
B(2,G)		$\boxed{\text{B(2,G)}}$								
$\boxed{\text{C(1,G)}}$										
D(∞)	→	D(∞)	→	D(∞)	→	D(∞)	→	D(∞)	→	D(14,E)
E(9,G)		E(9,G)		E(9,G)		E(9,G)		$\boxed{\text{E(9,G)}}$		
F(6,G)		F(6,G)		F(6,G)		$\boxed{\text{F(6,G)}}$				
Z(∞)		Z(14,C)		Z(14,C)		Z(14,C)		Z(14,C)		$\boxed{\text{Z(13,E)}}$

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: „Hill climbing“

- Informed add-on to **depth first search**:
- Among the possible candidates, expand the node with best heuristic value.
- In case of backtracking expand the next best node respectively.

Main problem: Long halt in local maxima

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: Optimistic hill climbing

- Special case of informed add-on to **depth first search**
- Expand only the node with best heuristic value.
- Backtracking is omitted: If heuristic value was wrong, the best result will not be found.

Main problem: Getting stuck in local maxima

Informed (Heuristic) Search Strategies

Given the following kind of information for weighted graphs:

Distance function h(state) being an *estimated* measure for the real distance to the target

- easily computable
- but accurate enough not to lead the search procedure to the wrong target

h() provides a nonnegative value: The smaller the value, the closer the target

Application: A* algorithm

- Informed add-on to **best first search**
- Expand the node where the sum of node label **plus** heuristic function is minimum.

Weitere Infos für die Anwendung von A* in öffentlichen Verkehrsnetzen:
Seminarvortrag und Ausarbeitung von Stefan Görlich, SS 2005, Nr. 5
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra's algorithm) *(State evaluation = Node evaluation)*

Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_B(u)$ for estimating the real distance $d_B(u)$ to target node B:

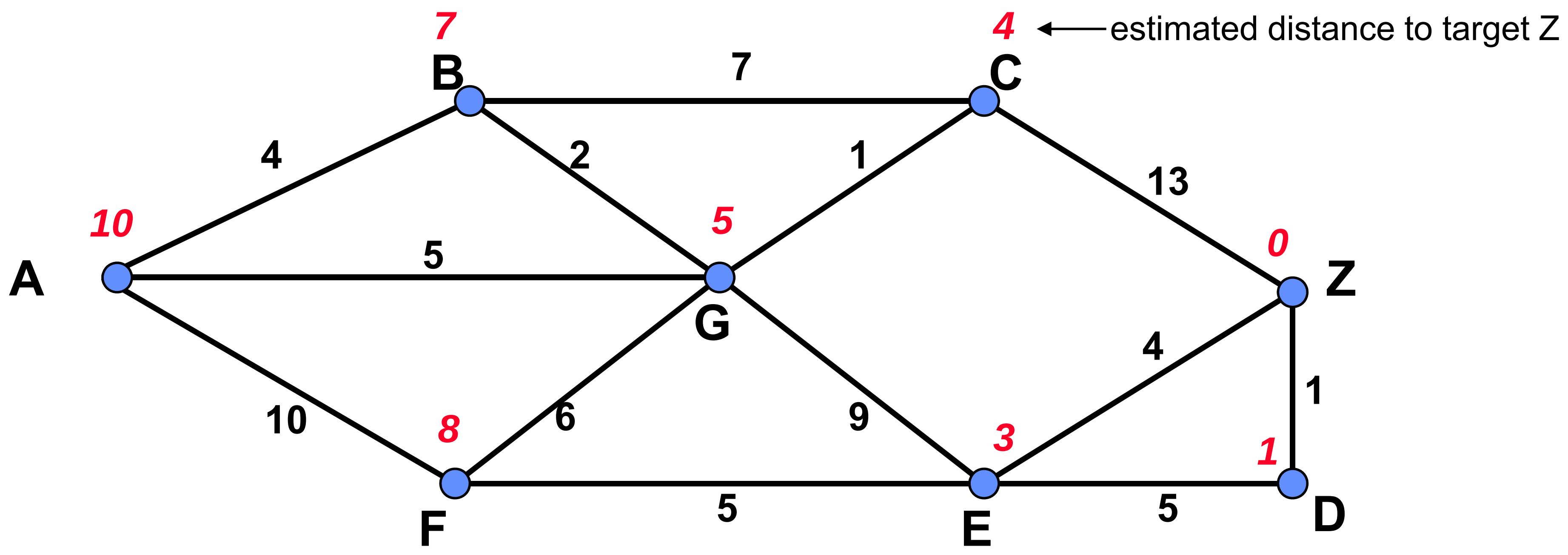
Admissibility: $h_B(u) \leq d_B(u)$

Monotonicity: $h_B(u) \leq h_B(v) + \mathit{length}(u,v)$

Algorithm for the search of a path from A to B having minimal global edge length:

- Put A into the set **`Done`**. Label A by *distance*(A) := 0.
  Put all other nodes into the set **`YetToCompute`**.
  Label all neighbors N of A by *distance* (N) := *length* (A,N) and
  heuristic (N) := *distance* (N) + $h_B(N)$
  and all other nodes by *distance* (V) := ∞ and *heuristic* (V) := ∞.
- Repeat:
  Choose node V from **`YetToCompute`** with minimum *heuristic* (V)
  and shift V to the set **`Done`**.
  Update all neighbors N of V that are still in **`YetToCompute`**:
  distance (N) := min {*distance* (N), *distance* (V) + *length* (V,N)}.
  heuristic (N) := *distance* (N) + $h_B(N)$ (if update is necessary).
  until V = B

Example for A* algorithm



Shortest path from G to Z: G → E → Z (13 units)

Node (real distance from G, direct predecessor, estimated distance to target):

A(5,G,15)		A(5,G,15)		A(5,G,15)		A(5,G,15)
B(2,G,9)		$\boxed{\text{B(2,G,9)}}$				
$\boxed{\text{C(1,G,5)}}$						
D(∞)	→	D(∞)	→	D(∞)	→	D(14,E,15)
E(9,G,12)		E(9,G,12)		$\boxed{\text{E(9,G,12)}}$		
F(6,G,13)		F(6,G,14)		F(6,G,14)		F(6,G,14)
Z(∞)		Z(14,C,14)		Z(14,C,14)		$\boxed{\text{Z(13,E,13)}}$

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra's algorithm)

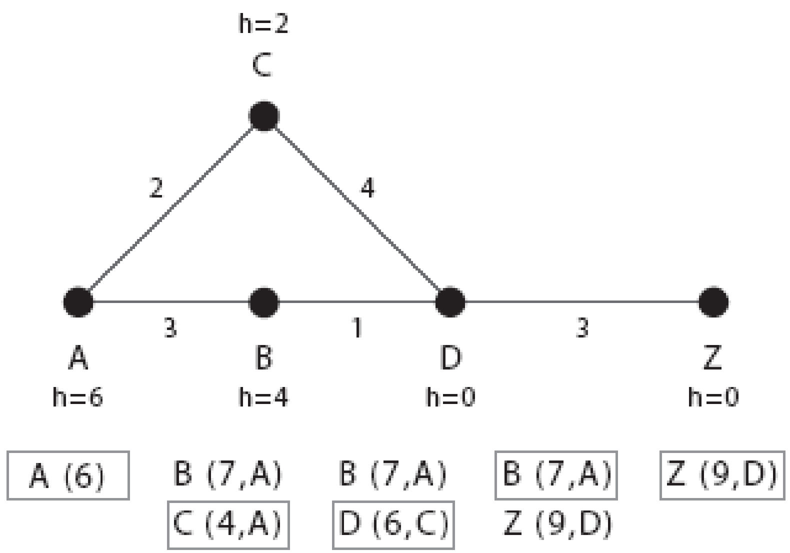
Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_B(u)$ for estimating the real distance $d_B(u)$ to target node B:

Admissability: $h_B(u) \leq d_B(u)$

What happens if monotonicity is abandoned ? $h_B(u) \leq h_B(v) + \mathit{length}(u,v)$

Example:



Aus: Diplomarbeit Andre Keller (SS 2008)

Error: D will not be updated anymore because it is already in **Done**

Informed (Heuristic) Search Strategies

A* algorithm for weighted graphs

(Generalisation of Dijkstra's algorithm) ***(State evaluation = Node evaluation)***

Requirement for edge weights: All edge lengths must be nonnegative.

Requirement for heuristic function $h_B(u)$ for estimating the real distance $d_B(u)$ to target node B:

Admissability only: $h_B(u) \leq d_B(u)$

Algorithm for the search of a path from A to B having minimal global edge length:

- Put A into the set **Done**. Label A by *distance*(A) := 0.
  Put all other nodes into the set **`YetToCompute`**.
  Label all neighbors N of A by *distance* (N) := *length* (A,N) and
  heuristic (N) := *distance* (N) + h_B(N)
  and all other nodes by *distance* (V) := ∞ and *heuristic* (V) := ∞.

- Repeat:
  Choose node V from **`YetToCompute`** with minimum *heuristic* (V)
  and shift V to the set **Done**.
  Update all neighbors N of V from **Done** and **`YetToCompute`**:
  distance (N) := min {*distance* (N), *distance* (V) + *length* (V,N)}.
  heuristic (N) := *distance* (N) + h_B(N) (if update is necessary).
  If an update occurred to a neighbor N* of **Done**: Shift N* back to YetToCompute
  until V = B

General Optimisation Methods for CSP

For the 1. search method introduced so far: Approaching global solutions via partial solutions:

Backtracking

- Test all constraints even if the variables are not all assigned
- States in which certain constraints are violated already should not be expanded further, but rather traced back.

Forward Checking

- Reduce all domains for variables not assigned such that the future assignment still has a chance to be feasible.
- Trace back if this leads to empty domains.

General Optimisation Methods for CSP

Example for forward checking:

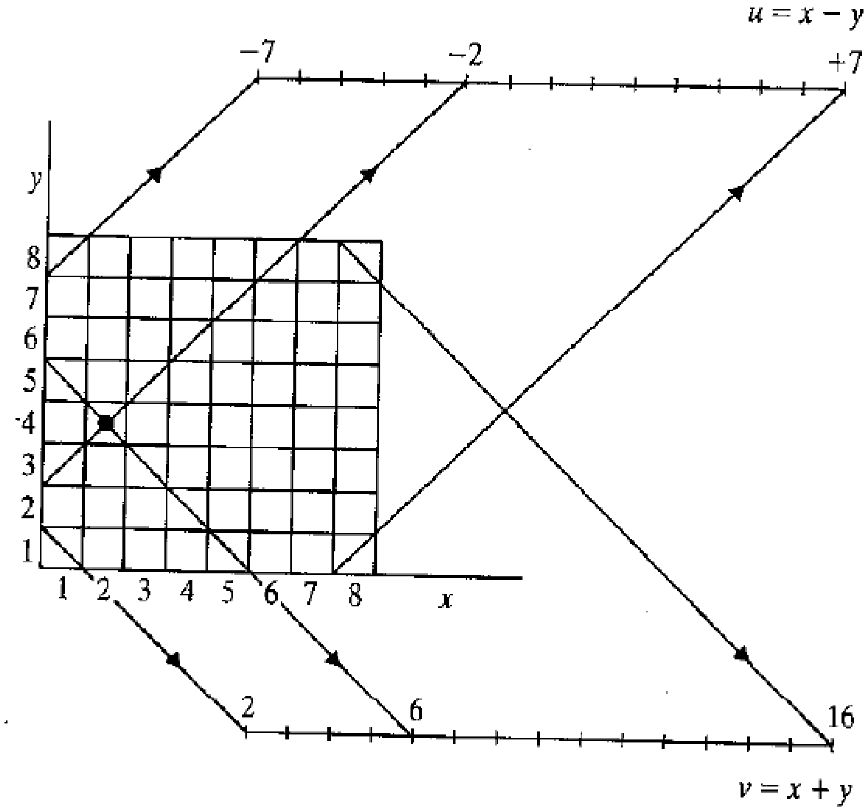
8-queens-problem (solution by Bratko, 3rd method)

Knowledge base:

```
queens3(YList) :-
   sol(YList, [1,2,3,4,5,6,7,8],
              [1,2,3,4,5,6,7,8],
              [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
              [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).

sol([],[], DomainY, DomainU, DomainV).

sol([Y |YTail], [X | XTail], DomainY, DomainU, DomainV) :-
   del(Y,DomainY,ReducedDomainY),
   U is X - Y,
   del(U,DomainU,ReducedDomainU),
   V is X + Y,
   del(V,DomainV,ReducedDomainV),
   sol(YTail, XTail, ReducedDomainY, ReducedDomainU,
ReducedDomainV).

del(Item, [Item|List], List).
del(Item, [First|Tail],[First|ResultTail]) :-
   del(Item,Tail,ResultTail).
```

Query:

```
?-queens3(YLIst).
```



Traversing search graphs

Alternative 2. search method:

Systematic improvement of preliminary (global) solutions

- **Node: describes state in search domain**
  - **State: Assignment of values to all variables (not all of them need be admissible)**
    Each state has got an evaluation.
- **Edge: Transition of a state into an adjacent state**
  (unsually feasible in both directions)
  - **Adjacent state: New values for certain variables keeping all values for the other variables**
- **Initial node: initial state**
  (is always unique)
  - **Initial node: Start with any assignment to the variables.**
- **Final node: final state wanted (problem solution)**
  (several ones are admissible)
  - **Final node: No adjacent state has got a better evaluation than the present one.**

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Idea:

- Start with an arbitrary assignment of values (valid or not).
- Assign new values for certain variables such that the new assignment bares fewer conflicts than the old one.

Advantages:

- happens to show good run time behaviour
- „repair strategy“ if something changes dynamically

Disadvantages:

- „Getting stuck“ in local minima
  - counter measures: random walk, tabu list, ...

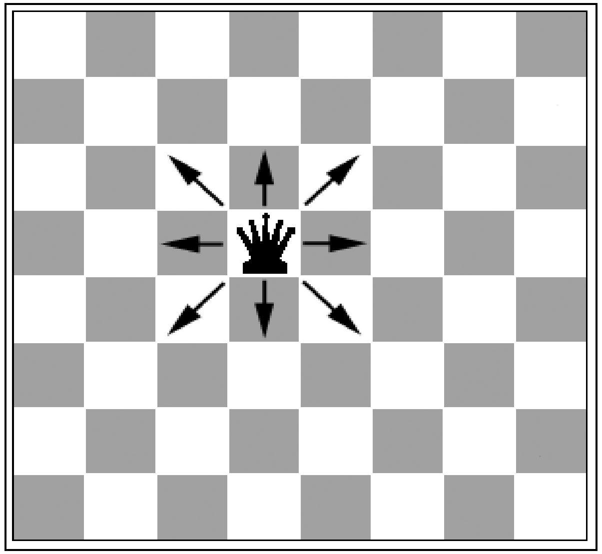
Weitere Details zum Thema Constraintsysteme:
Seminarvortrag und Ausarbeitung von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



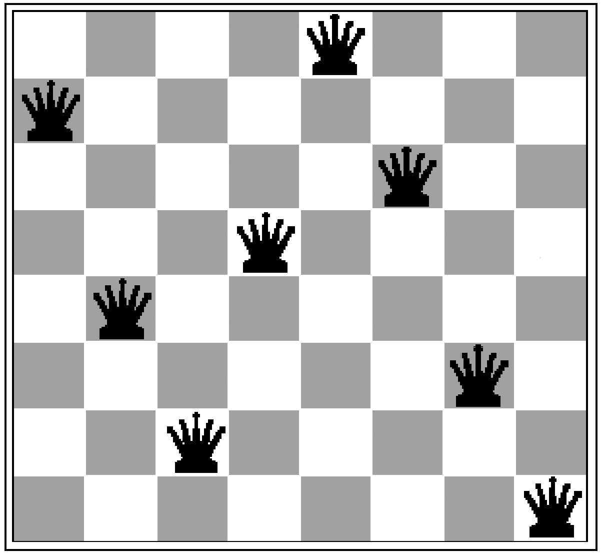
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



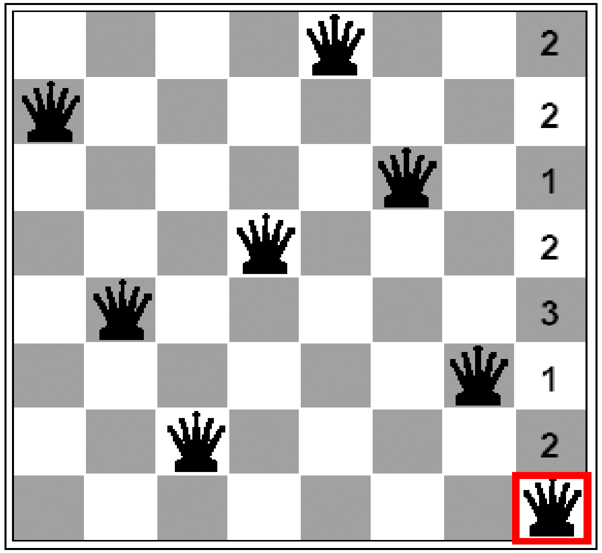
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



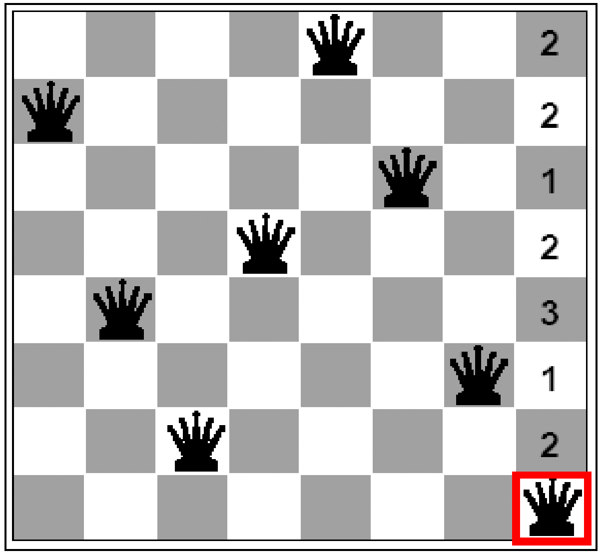
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

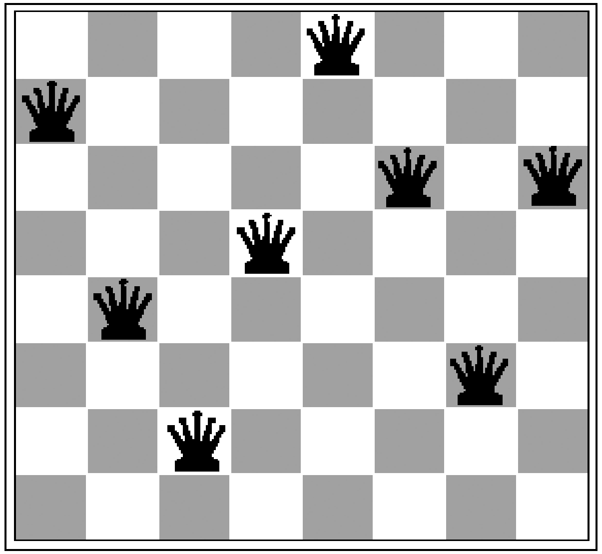
General Optimisation Methods for CSP

<u>For the 2. search method of systematic improvement</u>:

<u>Min-Conflicts procedure:</u>

Application: 8-queens-problem





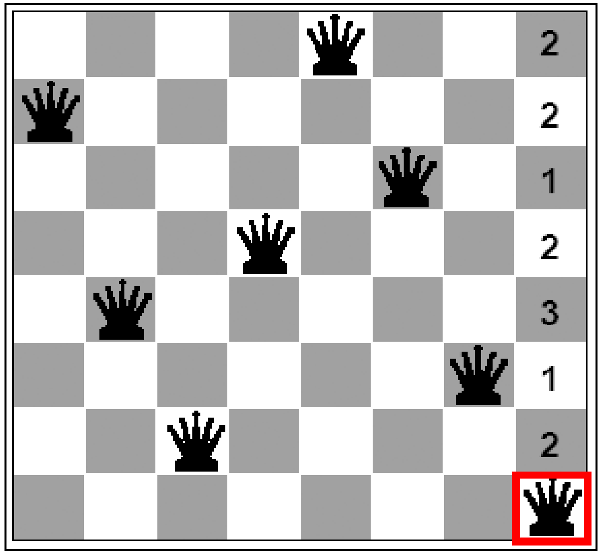
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

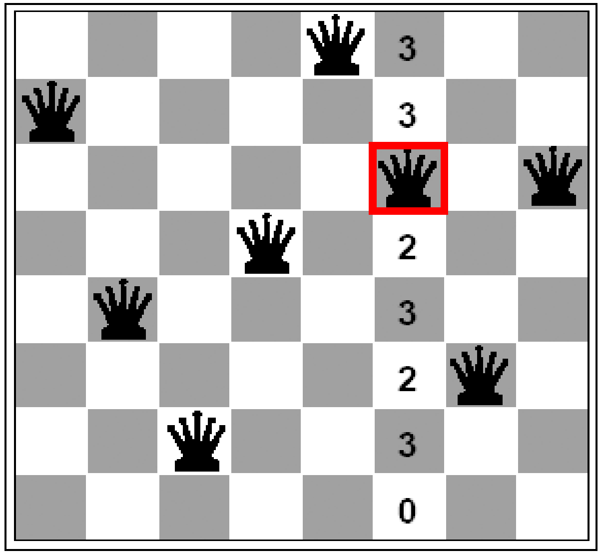
General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem





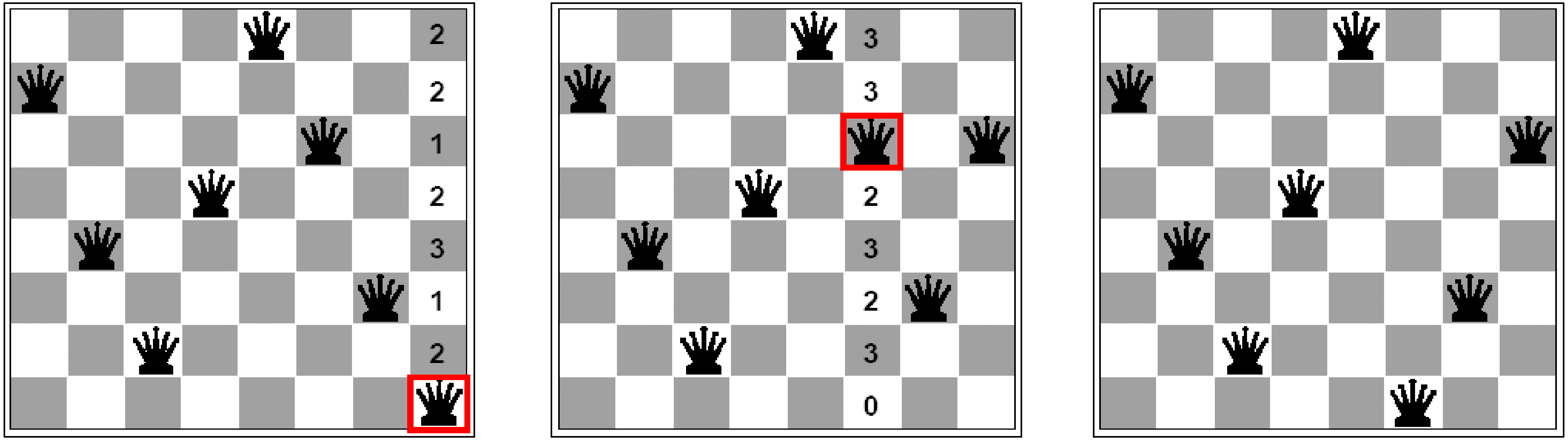
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Min-Conflicts procedure:

Application: 8-queens-problem



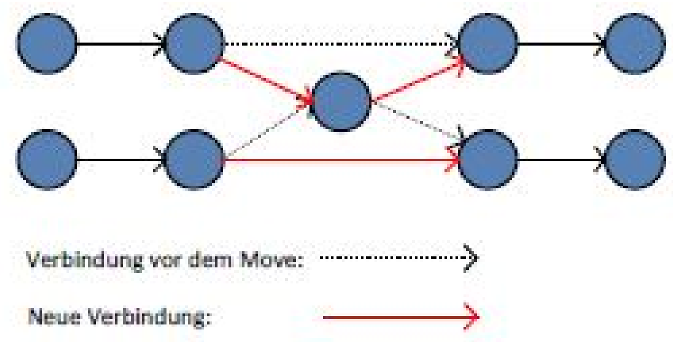
Quelle: Seminarvortrag von Stefan Schmidt, SS 2005, Nr. 6,
http://www.fh-wedel.de/archiv/iw/Lehrveranstaltungen/SS2005/SeminarKI.html

General Optimisation Methods for CSP

For the 2. search method of systematic improvement:

Working with tabu lists in search graphs:

- Determine a certain validity range for the algorithm, e.g. by a given number of operations
- Protocol all edges used in a transition from one state to another
- All edges used within the previous valisity range are not to be used again, neither their counterdirection.



Further enhancement: Simulated annealing

- Admit temporary deteriorations.
- Diminish the tolerance bound for deterioration in the course of algorithmic progress gradually.

These methods will mainly be used in improvements of global solutions

- Good results in logistics (TSP generalisations)